An Approach to Distributed State Space Exploration for Coloured Petri Nets*

L. M. Kristensen[1]** and L. Petrucci[2]

[1] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK
lmkristensen@daimi.au.dk
[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
petrucci@lipn.univ-paris13.fr

Abstract. We present an approach and associated computer tool support for conducting distributed state space exploration for Coloured Petri Nets (CPNs). The distributed state space exploration is based on the introduction of a coordinating process and a number of worker processes. The worker processes are responsible for the storage of states and the computation of successor states. The coordinator process is responsible for the distribution of states and termination detection. A main virtue of our approach is that it can be directly implemented in the existing single-threaded framework of Design/CPN and CPN Tools. This makes the distributed state space exploration and analysis largely transparent to the analyst. We illustrate the use of the developed tool on an example.

1 Introduction

State space exploration is one of the main approaches to computer-aided validation and verification [4, 7]. The basic idea of state space exploration is to compute all reachable states and state changes of the system and representing these as a directed graph. The main advantage of such exploration methods is that they are highly automatic to use and allow for investigating of many properties of the system under consideration. The main disadvantage of state space exploration methods is the *state space explosion problem* [17].

A wide variety of methods (see [17] for a survey) have been suggested in the literature to alleviate the state space explosion problem. Recently [5, 6], there has also been increased interest in exploiting the memory and computing power of several machines to conduct distributed state space exploration. Distributed exploration does not alleviate the state space explosion problem, but it increases the memory available for storage of the state space, and has the potential for a

* This work was started when both authors were at LSV, CNRS UMR 8643, ENS de Cachan, France.

** Supported by the Danish Natural Science Research Council.

linear speed-up in time. Distributed state space exploration has been developed, e.g., for the SPIN [1], UPPAAL [2], and Murϕ [16] tools.

In this paper we consider distributed state space exploration for Coloured Petri Nets (CPNs) [14]. Modelling and analysis of CPN models are supported by Design/CPN [9] and CPN Tools [8]. Until now, only very limited investigations have been conducted on distributed state space exploration for CPNs [11]. The contribution of this paper is to explore the use of distributed state space exploration in the context of CPNs and their associated computer tools Design/CPN and CPN Tools. A main requirement in the developent of our approach has been to exploit as much as possible the existing support for state spaces, and to make the distributed state space exploration largely transparent to the analyst.

The rest of this paper is organised as follows. Section 2 introduces some basic notations for state spaces. Section 3 presents our algorithm for distributed state space exploration and section 4 gives some experimental results with this algorithm on an example. Finally, section 5 contains the conclusions.

2 Background

Figure 1 lists the standard algorithm for sequential explicit state space exploration. The algorithm operates on two sets: UNPROCESSED which is a set of states for which successor states have not yet been calculated and NODES which is the set of already visited states. The algorithm starts from the initial state M_0 and conducts a loop until the set of unprocessed states is empty. In each iteration of the loop (lines 3–11), a state M is selected from the set of unprocessed states and the successor states M' of M are examined in turn. Successor states that have not been previously visited are inserted into the set of unprocessed states.

```
 1: UNPROCESSED ← {M0}
 2: NODES ← {M0}
 3: while ¬ UNPROCESSED.EMPTY() do
 4:    M ← UNPROCESSED.GETNEXTELEMENT()
 5:    for all ((t, b), M') such that M[(t, b)⟩M' do
 6:       if ¬(NODES.CONTAINS(M')) then
 7:          NODES.ADD(M')
 8:          UNPROCESSED.ADD(M')
 9:       end if
10:    end for
11: end while
```

Fig. 1. Sequential state space exploration algorithm.

The time taken to explore the state space of a CPN is determined by the computation of enabled binding elements and corresponding successor states in line 5, and the time used to determine whether a newly generated state has already

been explored before in line 6. This computation can be costly for large CPN models having many tokens and complex arc expressions. The space used to store the state space is another main factor in the algorithm. A distributed computation of the state space could potentially achieve the following: 1) Computation of successor states could be done in parallel for several states at a time, 2) Determining whether a state has already been explored could be done in parallel for several states at a time, and 3) the total amount of memory available for the state space exploration would be increased by having several machines.

3 Distributed State Space Exploration

Based on previous work [3, 15, 16] on distributed state space exploration, we choose to distribute both storage and calculation of successor states to a number of *worker processes*. The alternative would have been to only distribute the storage of states and compute sucessor states centrally. Distribution of successor states was considered of particular importance for CPNs, where the computation of successor states can be costly in case of many tokens on places and/or complex arc inscriptions. Unlike the approaches reported in [15], we introduce a central *coordinator process*. The purpose of the coordinator process is to distribute states. The introduction of the coordinator process makes the implementation of distributed state space exploration simpler given the single-threaded nature of Design/CPN and CPN Tools. Furthermore, the introduction of the coordinator process simplifies the termination detection. A third advantage of this architecture is that the coordinator can also be used later to control the verification process while the user interacts with the coordinator process only.

The basic idea is that when a worker computes a successsor state, it first checks if it itself is responsible for the state. Determing this is based on the use of an external hash function known to all workers and the coordinator process. If so, it checks locally whether the state is a new one. Otherwise, it sends the state to the coordinator. The coordinator then sends the state to the worker responsible for the state. The main disadvantage of this architecture is that the central coordinator node could become a bottleneck. On the other hand, the computation that needs to be done by the coordinator is very limited since it only consists in relaying states to the appropriate worker process.

The worker and the coordinator processes are all SML/NJ processes (Standard ML of New Jersey), and the communication infrastructure between the nodes is based on the Comms/CPN library [12]. This library supports communication between SML/NJ processes and external applications. In this particular case, all communications will be between SML/NJ processes. A main problem with the SML/NJ processes is that they are single-threaded and Comms/CPN only supports a blocking receive primitive. We have therefore added a CANRECEIVE primitive to Comms/CPN to support a polling receive. We assume that each of the workers will run on machines with equal computing power and communication between workers and coordinator will be on a local area network.

```
 1: computed ← false
 2:
 3: SEND(STATE M_0,worker(h_ext(M_0)))
 4: nextprobe ← h_ext(M_0)
 5: SEND(PROBE,worker(nextprobe))
 6: nextprobe ← nextprobe +1
 7:
 8: while ¬ computed do
 9:   for all i ∈ {1,...,n} do
10:     if CANRECEIVE(worker(i)) then
11:       RECEIVE(message,worker(i))
12:       if message == STATE M then
13:         nextprobe ← MIN(nextprobe,h_ext(M))
14:         SEND(STATE M,worker(h_ext(M)))
15:       else
16:         {Probe was returned}
17:         if nextprobe > n then
18:           computed ← true
19:         else
20:           SEND(PROBE,worker(nextprobe))
21:           nextprobe ← nextprobe +1
22:         end if
23:       end if
24:     end if
25:   end for
26: end while
27:
28: for all i ∈ {1,...,n} do
29:   SEND(STOP,worker(i))
30: end for
```

Fig. 2. State space exploration algorithm for the coordinator.

Figure 2 gives the algorithm executed by the coordinator process during the distributed state space exploration. The coordinator starts by sending the initial state M_0 to the worker determined by the external hash function $h_{\mathsf{ext}} : \mathbb{M} \to \{1, 2, \ldots, n\}$ used to distribute states between the n workers. It then sends a PROBE *message* to this process to be able to detect when the process has finished processing the state just sent. The PROBE messages and variable *nextprobe* are used to detect termination. We will explain how the termination detection works after presenting the algorithm for the workers. The coordinator then runs a loop where each of the worker processes is polled for messages using the CANRECEIVE primitive. If the received message is a STATE, this state is sent to the appropriate worker process and *nextprobe* is updated accordingly. If a PROBE message is received from a worker process, then it is passed on to the next worker and *nextprobe* updated accordingly.

Figure 3 lists the algorithm executed by each of the workers. The workers run in a loop exploring states received from the coordinator. Each worker will

```
1: stop ← false
2:
3: while ¬ stop do
4:   RECEIVE(message)
5:   if message == STATE M then
6:     if ¬NODES.CONTAINS(M) then
7:       NODES.ADD(M)
8:       UNPROCESSED ← {M}
9:
10:      while ¬ UNPROCESSED.EMPTY() do
11:        M ← UNPROCESSED.GETNEXTELEMENT()
12:        for all ((t, b), M') such that M[(t, b)⟩M' do
13:          if h_ext(M') ≠ i then
14:            SEND(STATE M')
15:          else if ¬NODES.CONTAINS(M') then
16:            NODES.ADD(M')
17:            UNPROCESSED.ADD(M')
18:          end if
19:        end for
20:      end while
21:    end if
22:  else if message == PROBE then
23:    SEND(PROBE)
24:  else
25:    stop ← true
26:  end if
27: end while
```

Fig. 3. State space exploration algorithm for worker i.

terminate once a STOP message is received from the coordinator. Whenever a state M is received, it is first checked if the state is already stored by the worker. If not, then it is added to UNPROCESSED and an exploration starting in M is conducted. States encountered in this exploration which belong to other workers are transmitted to the coordinator process, whereas encountered states that belong to the worker but are not currently stored are added to UNPROCESSED. When the exploration of the state space from the received state terminates, the worker goes back waiting for the next message. If a PROBE message is read, this message is sent back to the coordinator, and in case of a STOP message from the coordinator, the worker will stop its exploration.

The basic idea to detect termination of the state space exploration is that the coordinator passes a PROBE message among the workers. This solution is inspired by the distributed deadlock detection in [10]. The coordinator keeps track, in the *nextprobe* variable, of the next worker to send the probe to. The idea is that workers with an identity strictly smaller than *nextprobe* are known to be blocked, i.e., they are waiting for an incoming message in line 4 of the algorithm in figure 3. Hence, when the coordinator sends a state to a worker

with an identity which is smaller than the current *nextprobe*, this worker will be become active and thus the coordinator decreases the value of *nextprobe* accordingly (see line 13 in figure 2). When the coordinator sends the probe to a worker (see line 20 in figure 2), it updates the *nextprobe* to be the next worker. Hence, if states are not given to workers with a lower identity before the probe is returned, the next worker will then be probed. When the coordinator receives the probe from the last worker (line 17 of the algorithm in figure 2), then all workers are known to be in the blocked state and hence the state space exploration has been completed.

The distributed state space exploration requires a hash function h_{ext} mapping states onto workers. Assuming that the workers are running on machines of equal computing power in terms of memory and CPU, then this function should achieve two goals. Firstly, it should distribute the states uniformly across the n workers. Secondly, it should ensure a certain degree of *locality*, i.e., to reduce the communication overhead we would like as many successors states of a given state to reside on the same machine. To some extent these are conflicting goals.

To achieve a degree of locality we can select a small subset of the places P_h of the CPN and let the hash function depend on the marking of these places. All transitions in the CPN model which do not have a place in P_h as input or output will hence not affect the hash value and when such a transition is enabled from a state M, the successor state M' will belong to the same machine as M. Intuitively, we can thus control the degree of locality by the size of the set P_h. On the other hand, the set of reachable markings of the set P_h must accommodate a preferably even distribution of states onto the n workers.

Another issue is the relationship between the external hash function h_{ext} and the internal hash function h_{int} used in the internal hash table on each of the workers. Here we assume that all workers will be using the same internal hash function. We need to ensure that these hash functions are independent, i.e., that the external hash function h_{ext} does not cause the internal hash function h_{int} to map the states stored on that worker into a very small set of values. One approach to avoid this is to let the internal hash function h_{int} depend on the marking of places in $P \setminus P_h$. It should however be mentioned that the marking of places in $P \setminus P_h$ could be related via place invariants to the marking of places in P_h, and hence the markings may not be totally independent. The issue of the external hash function will be investigated further in section 4.

4 An Example

We now present a set of initial results obtained with the basic state exploration algorithm presented in the previous sections. We consider the distributed database system from [14] in figure 4. The CPN model describes the communication between a set of database managers $D = \{d_1, d_2, \ldots, d_N\}$ for maintaining consistent copies of a database in a distributed system. The idea of the protocol is that when a database manager updates its local copy of the database, requests are sent to the other database managers for updating their copy of the database.

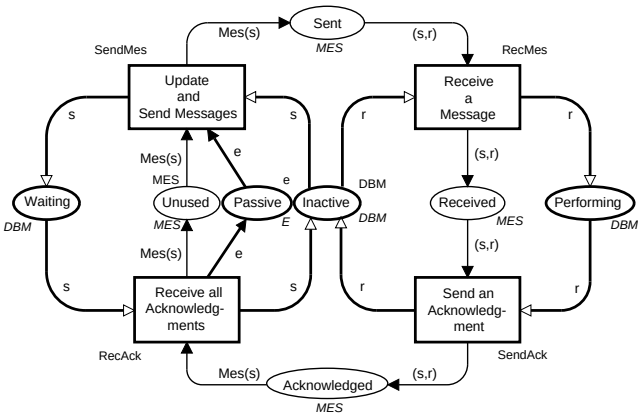

Fig. 4. The distributed database example.

When each database manager has updated its copy, it sends an acknowledgement back to the initiating database manager to confirm that the update has now been performed. A database manager in the protocol can either be in a state Waiting (for acknowledgement), Performing (an update requested by another database manager) or Inactive after having sent the acknowledgement back. All database managers are initially Inactive.

Table 1 gives some statistics for the distributed data base system where we have used 1–10 workers. For these experiments, the external hash function h_{ext} is based on the standard internal hash function h_{int} used by Design/CPN:

$$h_{\mathsf{ext}}(M) = (h_{\mathsf{int}}(M) \bmod n) + 1 \tag{1}$$

The $|DBM|$ column gives the number of database managers considered and the n column specifies the number of workers. *States* gives the total number of states in the state space and *Time* specifies the total time used for the exploration as measured in the coordinator process. The exploration time is written on the form *mm*:*ss* where *mm* is minutes and *ss* is seconds. The *Transmit* column specifies the total number of states transmitted. *External* indicates the number of successor states computed that were external while *Internal* gives the number of successor states computed that were internal. This is measured across all the workers. Column *Stored* gives the total number of states received by the worker which were already stored by the worker when received.

Table 2 gives detailed statistics for the workers. The $|DBM|$ column gives the number of managers considered and the n column specifies the number of workers. For each worker $W1$–$W10$, we list the number of states stored on the worker and in Table 3 the total time that the worker spent in the blocking state.

To reduce the number of states transmitted, a small cache is introduced on each process. This cache contains a set of recently sent states and is consulted before transmitting a state. The best results obtained in our experiments with

Table 1. Initial experimental results for the database system.

|DBM|	n	States	Time	Transmit	External	Internal	Stored
10	1	196,831	112:49	1	0	1,181,000	0
10	2	196,831	58:25	590,501	590,500	590,500	585,380
10	3	196,831	63:00	1,180,981	1,180,980	20	984,160
10	4	196,831	83:03	1,180,991	1,180,990	10	984,160
10	5	196,831	108:59	1,181,001	1,181,000	0	984,170
10	6	196,831	47:38	1,180,991	1,180,990	10	984,160
10	7	196,831	40:48	590,511	590,510	590,490	492,090
10	8	196,831	27:12	1,180,991	1,180,990	10	984,160
10	9	196,831	57:08	1,181,001	1,181,000	0	984,170
10	10	196,831	77:07	1,181,001	1,181,000	0	984,170

Table 2. Number of nodes on workers for the database system.

|DBM|	n	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
10	1	196,831									
10	2	98,410	98,421								
10	3	65,610	65,610	65,611							
10	4	49,210	49,211	49,200	49,210						
10	5	40,330	47,521	25,280	53,940	29,760					
10	6	31,950	33,660	34,510	33,660	31,950	31,101				
10	7	6,720	46,090	20,160	23,221	40,320	6,560	53,760			
10	8	24,130	24,240	24,600	24,240	25,080	24,971	24,600	24,970		
10	9	12,960	29,160	17,101	23,490	23,490	17,100	29,160	12,960	31,410	
10	10	20,160	23,761	12,640	26,970	14,880	20,170	23,760	12,640	26,970	14,880

the database example uses a cache with 200 states. Hence, this configuration is used hereafter.

Until now we have used the internal hash function in each worker as a basis for the external hash function. For the DBM system it is however possible to come up with a tailored hash function based on the observation that except for the initial marking only one of the database managers is active at a time. If we let $\mathit{Waiting}(M)$ denote the database manager which is waiting (if any) in the state M and 1 if there is none, then a possible external hash function for the DBM system would be:

$$h_{\mathsf{ext}}(M) = (\mathit{Waiting}(M) \bmod N) + 1 \tag{2}$$

With this hash function it only makes sense to have at most N workers. The experiments conducted with 10 database managers and a cache size of 200 states are presented in Table 4. The hash function splits the state space into 9 sets of 19,683 states each, plus one containing also the initial marking (19,684 states). These sets are evenly distributed among the computation nodes. In these experi-

Table 3. Waiting time of workers for the database system.

|DBM|	n	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
10	1	0:00									
10	2	4:22	8:33								
10	3	23:43	23:38	18:40							
10	4	43:52	21:57	29:08	55:24						
10	5	27:22	40:50	80:46	64:01	99:50					
10	6	13:30	10:34	10:15	15:49	33:46	32:39				
10	7	40:03	7:10	35:10	35:12	25:10	40:16	1:07			
10	8	12:43	5:36	7:20	15:07	7:11	11:57	6:57	10:44		
10	9	54:21	37:40	51:23	48:07	49:24	52:04	39:44	54:23	9:53	
10	10	42:23	51:36	67:18	60:19	73:56	52:49	51:33	68:36	14:01	64:17

ments 18 of the new states computed are external while the 1,180,982 others are internal. Hence, there is very little communication between the processes and they can perform their local computations without waiting for other workers to provide new states to handle. The results obtained are thus considerably better than with the original hash function. For 9 and 8 workers, some computation nodes are handling 2 groups of states, which explains the longer computation time, and the longer blocking time for the other workers.

Table 4. Experimental results using the tailored hash function.

n	Time	Block. Time
10	3:39	2:14
9	5:48	4:22
8	5:58	4:33

5 Conclusions and Future Work

We have described an approach to conducting distributed state space exploration for CPNs and presented some experimental results obatined with an implementation of our approach within Design/CPN. The experimental results are encouraging and indicate that the art of distributed state space exploration is in the choice of a good external hash function mapping states onto workers. In the general case, we would like to automatically derive an external hash function without relying on the user knowledge of the system to specify a good hash function. Hence, we need a way of determining a set of places that can be used to obtain a good hash function. A possible approach to this is to conduct an initial partial state space exploration (depth-first and breadth-first) and from the markings encountered attempt to derive a hash function. This hash function can then

be based on counting the tokens on a selected set of places, or eventually, when dealing with places having simple colour sets such as integers or enumerations, on the rank of the colours in the place. Future work includes exploring more elaborated approaches to perform model-checking in our distributed framework. A possible starting point for this work would be [13, 15].

References

1. J. Barnat, L. Brim, and J. Stříbrná. Distrubuted LTL model checking in SPIN. In *Proc. of SPIN 2001*, volume 2057 of *LNCS*, pages 200–216. Springer-Verlag, 2001.
2. G. Behrmann. A Performance Study of Distributed Timed Automata Reachability Analysis. In Lubos Brim and Orna Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier, 2002.
3. G. Behrmann, T. Hune, and F. Vaandrager. Distributed Timed Model Checking - How the Search Order Matters. In *Proc. of CAV'00*, volume 1855 of *LNCS*, pages 216–231. Springer-Verlag, 2000.
4. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer-Verlag, 2001.
5. L. Brim and O. Grumberg, editors. *Proc. of 1st Workshop on Parallel and Distributed Model Checking*, volume 68 of *Electronic Notes in Theoretical Computer Science*, October 2002.
6. L. Brim and O. Grumberg, editors. *Proc. of 2nd Workshop on Parallel and Distributed Model Checking*, volume 89 of *Electronic Notes in Theoretical Computer Science*, September 2003.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.
8. The CPN Tools Homepage. `http://www.daimi.au.dk/CPNtools`.
9. The Design/CPN Homepage. `http://www.daimi.au.dk/designCPN`.
10. E.W. Dijkstra. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, 16:217–219, 1983.
11. G. Farret and G. Carré. Distributed Methods for Computation and Analysis of a Coloured Petri Net State Space. Master's thesis, Department of Computer Science, University of Aarhus, August 2002.
12. G. Gallasch and L. M. Kristensen. Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of the 3rd Workshop on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pages 79–93. Department of Computer Science, University of Aarhus, 2001. DAIMI PB-554, ISSN 0105-8517.
13. H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *Proc. of SPIN 2001*, volume 2057 of *LNCS*, pages 217–234. Springer-Verlag, 2001.
14. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 1: Basic Concepts.* Springer-Verlag, 1992.
15. F. Lerda and R. Sisto. Distributed-Memory Model-Checking with SPIN. In *Proc. of SPIN 1999*, volume 1680 of *LNCS*, pages 22–39. Springer-Verlag, 1999.
16. U. Stern and D.L. Dill. Parallelizing the Murϕ Verifier. In *Prooceedings of CAV'97*, volume 1254 of *LNCS*, pages 256–278. Springer-Verlag, 1997.
17. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.